LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-685714

The Near Zone to Far Zone Transformation (N2F)

D. T. Blackfield, B. R. Poole

March 11, 2016

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

The Near Zone to Far Zone Transformation (N2F)

Donald Blackfield and Brian Poole, LLNL[1]

April 16, 2013

Brief description of near zone to far zone transformation[2]

N2F is a C/C++ code used to calculate the far zone electromagnetic (EM) field, given E and H near zone field data. The method used by N2F can be found in Ref. 1 and 2. N2F determines the far field E_ϕ and E_θ in spherical coordinates for near zone data calculated in either Cartesian or Cylindrical geometry.

Using the methodology in Refs. 1 and 2, N2F calculates

$$E_\theta = -\eta W_\theta - U_\phi \tag{1}$$

and

$$E_\phi = -\eta W_\phi + U_\theta \tag{2}$$

where η is the free space impedance

$$\eta = \mu_0 c \tag{3}$$

with $\mu_0 = 4*\pi x10^{-7}$ (H/m) and c is the speed of light.

$$W(t) = \frac{1}{4\pi Rc}\frac{\partial}{\partial t}\left\{\int_{s'} J_S(t + (\bar{r}'.\hat{r})/c - R/c)ds'\right\} \tag{4}$$

$$U(t) = \frac{1}{4\pi Rc}\frac{\partial}{\partial t}\left\{\int_{s'} M_S(t + (\bar{r}'.\hat{r})/c - R/c)ds'\right\} \tag{5}$$

where

$$J_S(t) = \hat{n} \times H(t) \text{ and } M_S(t) = -\hat{n} \times E(t) \tag{6}$$

N2F calculates the retarded or time delay, at each near zone point in Cartesian coordinates where

$$\text{time delay} = (\bar{r}'.\hat{r})/c \tag{7}$$

---

[1] This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

[2] In this document, words in *italics* are code variable names**, bold** are function names, **bold underline** are file names (either input or output) and ***bold italics*** UNIX internal functions.

This value is added to the input time for each near zone point to translate the J and M time arguments in Equations 4-5 to a global reference time grid with $(\bar{r}'.\hat{r})$ the distance from the near zone point to the far zone point. In Equations 4-5 $\hat{r}$ is the unit vector along the path to the far zone point, $\bar{r}'$ is the vector to the center of the near zone cell to be integrated and S' in Eq. 4-5 the surface area of integration. Typically, the 1/R term outside of the integrals is set to 1 and the constant R/c, representing a simple phase shift suppressed (see Ref 1 pp. 108-109). Figure 1 is a diagram of the Cartesian geometry used for the integrations in Equations 4-5.

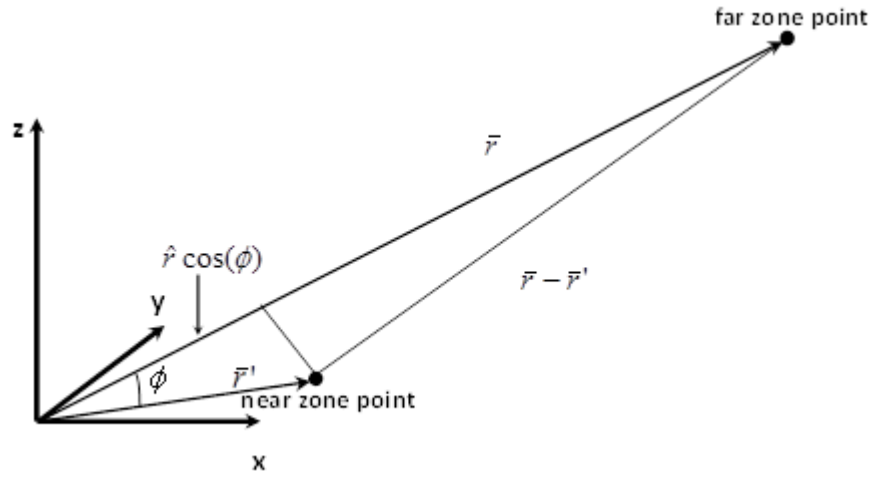

Figure 1. Diagram of the distance from a near zone point to the far zone point in Cartesian geometry. In Equations 4-5, $\hat{r}$ is the unit vector along the path to the far zone point, $\bar{r}'$ is the vector to the center of the near zone cell to be integrated.

Brief Overview of Code Logic

The user must supply the near zone locations together with the corresponding near zone E and H data, and set the input geometry flag. For cylindrical geometry, **Gz1.flt** contains the bottom of the near zone cylinder geometry data in IEEE 32-bit floating point (x, y, z) triplets. **Gz2.flt** contains the near zone cylinder top geometry data using triplets while Gr1.flt contains the near zone cylinder side triplet geometry data. **Fz1.flt, Fz2.flt** and **Fr1.flt** contain the corresponding E and H t field data stored in groups of 7 IEEE 32-bit floating point data for each geometry point (t, E_r, E_ϕ, E_z, H_r, H_ϕ, H_z). For Cartesian near zone data, **Gx1.flt**, **Gx2.flt**, **Gy1.flt**, **Gy2.flt**, **Gz1.flt**, and **Gz2.flt** contain the IEEE 32-bit floating point triplet near zone geometry data (x, y, z) for the –x, +x, -y, -y, -z, and +z surfaces respectively. The corresponding E and H field data using the 7 IEEE 32-bit floating point element grouping (t, E_x, E_y, E_z, H_x, H_y, H_z) are stored in the files **Fx1.flt, Fx2.flt**, **Fy1.flt**, **Fy2.flt**, **Fz1.flt**, and **Fz2.flt** respectively. N2F determines the size of the problem after calculating the total number of near zone points, including the additional ϕ cylindrical coordinates multiplied by the number of time points. Depending upon the size of available memory and the total amount of data, either a large or small problem computational architecture is used. For small problems, the entire near zone data set is loaded into memory. For large

problems, the near field E and H data is loaded into memory at each near zone point as needed, with a resulting slowing of the calculation. A warning message is sent when N2F determines that the size of the problem is large. Different code logic is used based on the geometry flag (*igeom* = 0 for Cartesian and 1 for cylindrical) and the size of the problem (*imem* = 0 for small and 1 for large).

What N2F does is replace the integrals with sums assuming J and M are uniform in each cell area containing a near zone cell centered point. The near zone E and H data which are assumed to be calculated on near zone point nodes are translated to cell centered values, with the input near zone nodes located at the cell corners. In cylindrical geometry, the near zone data is collected on lines in (r, z) space. The ϕ values are obtained by replicating the near zone geometry (r, z) and the near zone E and H field data assuming ϕ symmetry. The near zone E and H data set is used, with ϕ only influencing distance from the near zone point to the far zone point or the time delay, the distance it takes an EM signal to travel from the near zone location (changing with ϕ) to the desired far zone location.

First N2F calculates all of the near zone cell areas and stores them in an array, converting from a geometry based on nodes to a grid where the data is cell centered. While this is straightforward in cylindrical geometry, for Cartesian geometry, near zone points on the edges and corners of the near zone data surface have different areas than those cell areas in the surface interior. N2F finds all of the near zone points that fall on either an edge or corner and prints the number of edge and corner points to the screen. N2F also prints the total area and the area of each surface to the screen.

Having found the cell areas, N2F then calculates the time delay from each near zone point, including replicated points in cylindrical geometry, to the desired far zone point and stores these values in an array. Next it takes the E and H near zone node data and translates it to cell centered data.

Then N2F, using the previously calculated and stored time delay, interpolates the E and H time data for each near zone point onto a common reference time grid using linear interpolation. Having translated the E and H data from cylindrical to Cartesian coordinates if required, the J and M values for each near zone point is calculated and integrals are performed using sums. Having now created integrals found in Equations 4 and 5 in Cartesian coordinates, the numerical time derivatives of these integrals are performed to calculate U(t) and W(t) using centered differencing. These values are then translated to spherical coordinates before calculating E_ϕ and E_θ found in Equations 1-2.

Input to NF2 version 1.001

The first argument selects the problem geometry (*igeom*) with 0 designating Cartesian and 1 cylindrical. The next two arguments are related to the far zone spherical angles, θ (azimuthal) and ϕ (polar) angles (see Fig 2). The second argument, θ, when equal to or less than zero is the azimuthal angle of the desired far zone location. This value is changed by N2F to a positive value in degrees. If this argument is initially

positive, then θ is the number of degrees in a computational slice on the azimuthal angle. The number of equally spaced azimuthal far zone calculations is given as

$$n_\theta = \textbf{\textit{int}}(180./\theta) \tag{8}$$

A similar formulation applies for the third argument ϕ, the polar angle in degrees. In this case when ϕ is less than or equal to zero, the far zone calculation is performed for a single polar angle, the absolute value of the third argument. If ϕ is greater than zero, then this input designates the slice polar angle and the number of polar angles used is

$$n_\phi = \textbf{\textit{int}}(360./\phi) \tag{9}$$

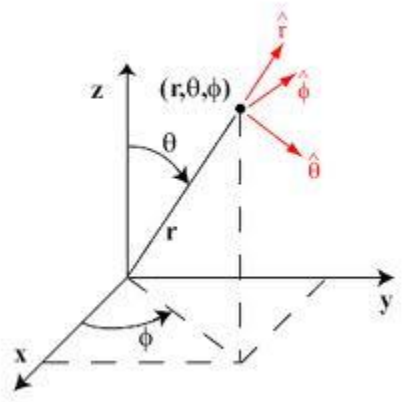

Figure 2. Spherical coordinates for the far zone calculation surface. Spherical coordinates are a system of curvilinear coordinates for describing positions on a sphere. Define ϕ to be the azimuthal angle in the x-y plane from the *x*-axis with $0 \le \phi \le 2*\pi$, θ to be the polar angle from the positive *z*-axis with $0 \le \theta \le \pi$, and r to be distance (radius) from a point to the origin. This is the convention commonly used in physics.

If the fourth argument is non-zero, then output files are created. The file names contain the format $eti_\theta pi_\phi$ where i_θ is the number of the azimuthal slice (from 1 to n_θ) and i_ϕ is the number of the polar slice (from 1 to n_ϕ). For example, the second azimuthal angle and the third polar angle data file would have the name **et2p3.txt**.

Finally, for cylindrical geometries, the input field data being symmetric in the cylindrical angle must be normalized to the number of polar angles used to calculate this input data. To obtain the correct far zone area normalization, the number of uniform polar angles used in the calculation of the input near zone field data (*nphi_data*) must be given to N2F. This is the fifth argument. N2F has a normalization variable

$$cnst = 4*\pi*c \tag{10}$$

where π is 3.141592653589793238 and c is the speed of light (2.99792458×10^8).

When cylindrical geometry is selected, *cnst* contains an additional normalization factor, related to the cylindrical cell area,

$$cnst = 4*\pi*c*nphi\_data \tag{11}$$

While a single set of input near zone data cylindrical data is read, symmetry allows us to "replicate" this data *nphi_data* times to obtain the correctly normalized far zone fields.

Default Input values

The following are the default input values.

$$\begin{aligned} igeom &= 0 \\ \theta &= 0.0 \\ \phi &= 0.0 \\ ifileout &= 0 \\ nphi\_data &= 1 \end{aligned} \tag{12}$$

When no arguments are supplied, the far zone calculation uses Cartesian geometry at a single far zone location, $\theta = 0$ and $\phi = 0$ and no output files are generated.

Creating the near zone geometry and field data files names

First, if *igeom* = 1, the cylindrical input data is sent to the terminal screen. The ϕ delta for which the input data is associated is calculated

$$dphi\_data = 180./(\textbf{\textit{double}})nphi\_data \tag{13}$$

This is required, because in cylindrical coordinates, only a single set of data is and then assumed to be replicated *nphi_data* times, assuming equally spaced ϕ angles.

For cylindrical data N2F expects near zone data with the following file names **Gz1.flt**, **Gz2.flt**, and **Gr1.flt** for the bottom, top, and side of the near zone cylindrical data. The corresponding field data files should be named **Fz1.flt**, **Fz2.flt**, and **Fr1.flt**.

For cylindrical coordinates, three distinct sets of near zone data are read. The first file (**Gz1.flt**) read corresponds to the bottom of the near zone cylindrical data. The second file (**Gz2.flt**) read is associated with the top of the near zone cylindrical data. The third file (**Gr1.flt**) corresponds to the data on a vertical line on the side of the cylinder.

These near zone data files are in (2 x 3 = 6) separate data files. Each field file has an associated geometry file. The geometry file has the r and z values of the near zone data, the cylindrical angle ϕ not required.

When *igeom* = 0, Cartesian near zone data is read. The near zone data lies on the surfaces of a six sided box, not necessarily a cube. The geometry data contains triplets (x, y, z) at each point on the near zone data surface. N2F expects near zone data with the following file names **Gx1.flt**, **Gx2.flt**, **Gy1.flt**, **Gy2.flt**, **Gz1.flt**, and **Gz2.ft**. The corresponding field data files should be named **Fx1.flt**, **Fx2.flt**, **Fy1.flt**, **Fy2.flt**, **Fz1.flt**, and **Fz2.flt**.

N2F assumes that the near zone data surrounds the origin, so the data on the y-z plane at x < 0 is contained in the geometry file **Gx1.flt** with the associated near zone field data in the file named **Fx1.flt**. The y-z plane data at x > 0 is found in input file **Gx2.flt** and the associated field data in **Fx2.flt**. The x-z plane geometry and field data for y < 0 is in **Gy1.flt** and ***Fy1.flt*** respectively while the x-z plane y> 0 data is in **Gy2.flt** and **Fy2.flt**. Finally the x-y plane geometry data for z < 0 is in ***Gz1.flt*** with the corresponding field data in **Fz1.flt** with the x-y plane data for z > 0 in the geometry file **Gz2.flt** and the associated field data in **Fz2.flt**.

When *igeom* = 0, a message appears on the terminal showing that Cartesian geometry will be used. For all problems, the input far zone θ and ϕ values as well as *ifileout* which controls the creation of output files is echoed to the screen. In addition, for the cylindrical geometry, *nphi_data*, the number of replication of the data in ϕ is also displayed.

Determining the number of far zone calculation points

The number of far zone points to be computed is found below

$$ncase = n_\theta * n_\phi \tag{14}$$

where n_θ and n_ϕ are calculated above, from the input values for θ and ϕ. If either the input value of θ and/or ϕ = 0, then n_θ and/or n_ϕ is set to 1.

Summary output file

The summary output file, always written, is created with the name **sum.txt**

Determine the number of near zone point on each surface

It should be noted, that the far zone calculation uses cell centering. However, N2F assumes that the input near zone geometry and field data is node centered. If cell centered input is used, the far zone calculations will not be correct. N2F also assumes that on each surface, all interior nodes are equally spaced, although the spacing on each surface may differ. The input data should contain nodes on a regular grid with the data entered into the input file in a regular, non-random order. For Cartesian data, the edge nodes while equally spaced may have a different spacing than interior points. The cells that use edge and/or corner nodes have their areas calculated differently than interior cells.

The function **get_npoints** determines the number of points in a surface file. When *igeom* =1, the number of cell centered points equals the number of node centered points minus 1. First, this function counts the number of bytes in a given geometry file.

Assuming that there are 4 bytes to a float, and that all the input files contain floats, then for cylindrical geometry the number of points (n) in a file that contains n_{bytes} of data

$$n = n_{bytes} / (4 * 2) \tag{15}$$

with cylindrical geometry files containing the doublets (r ,z).

In Cartesian coordinates, the geometry files contain triplets (x, y, z) so the number of points (n) in a file containing n_{bytes} of data is

$$n = n_{bytes} / (4 * 3) \tag{16}$$

The function **get_npoints** returns the integer n.

The total number of input points is stored in *npmax*. For cylindrical geometry, taking into account the replication in ϕ of the input data, the total number of points stored

$$npmax\_nphi = npmax * nphi\_data \tag{17}$$

where *nphi_data* = 1 for Cartesian geometry.

Determine number of time steps

The input near field data is time dependent. N2F determines the number of time steps in the input data, assuming that the input data is at uniform time steps. For non-uniform time steps, the input data should be interpolated onto a uniform time line before being used for far zone calculations. The number of time steps *ndata* is determined in the function **get_ndata**. The near zone E and H field file (**Fz1.flt** for cylindrical problems or **Fx1.flt** for Cartesian ones) is opened and the number of bytes counted to determine the number of floats in the file. N2F assumes that all the E and H near zone data contain the same number of time points. Next, the total number of coordinate points (*npoints*) having been passed is multiplied by 7 so that the number of points *ndata*

$$ndata = n_{bytes} / (7*npoints) \tag{18}$$

Each field file contains groups of 7 floats. Each group consists of a time point followed by 3 components of the E field points followed by 3 components of the H field points for a total of 7 floats. Note that for cylindrical data, *npoints* was decreased by 1 to translate from node to cell centered data. The function **get_ndata** adds 1 to the *npoints* to translate back to node data when determining the total number of time steps. The function **get_ndata** returns the total number of time steps.

Determine the maximum problem size before swapping occurs

The input field data sets can easily become quite large, especially when there are many time steps. N2F uses logic dependent upon the size of the input data. The function **get_memsize** determines the limit on the product of the total number of points times the number of time steps based upon the amount of free memory available. This uses the UNIX system call ***free*** with the ***–b*** option to obtain the amount of free memory in bytes. This number is written to a temporary file in the execution directory. After this number is read, the file is deleted. Currently small problems have

$$npmax\_nphi * ndata * MEMFACTOR \leq memsize \tag{18}$$

where *memsize* = amount of free memory in bytes and *MEMFACTOR* is the large problem conversion factor translating the problem size into bytes. For an 8 GB UNIX computer, this factor is (400.0/8.55). For running on other UNIX platforms, a problem should be run a few times varying number of time steps. Using the UNIX top command in one window while running N2F in another, the appropriate *MEMFACTOR* value can be obtained. Note that this factor is approximate since there is some overhead

which does not scale with the number of time points. For sufficiently large problems, the percentage of memory used by this overhead should be small.

Dynamically created arrays for "small" problems

When the memory requirement is small, *imem* = 0, otherwise it is 1. When *imem* = 1, a warning message is sent to the screen. In either case the total number of points times the number of time steps and the amount of free memory required and available are printed to the screen.

When *imem* = 0, a large dynamically created array is used to store all of the time delays for all far zone points. Otherwise these delays are calculated several times during the simulation with a resulting slowing of the calculations. For small problems, when *igeom* = 0 (Cartesian) all of the delays (for each input geometry data point for each distinct output θ and ϕ combination) are stored in a large 2D array (*npmax* X *ncase*) named *timedelay* where

$$ncase = n_\theta * n_\phi \tag{19}$$

This array is dynamically created and initialized to zero.

Next, for both large and small problems, a 1D array to hold the *npmax* cell areas is dynamically created and initialized to zero. If the number of near zone geometry points is too large, the dynamic memory allocation will fail and N2F will print an error message to the screen before terminating.

For cylindrical geometry (*igeom*=1), cos(ϕ) and sin(ϕ) for each value of ϕ to be calculated is stored in the dynamically allocated and zero initialized cosp and sinp arrays respectively.

For Cartesian geometry, all of the near zone x data is store in the dynamically allocated and initialized to zero array *xsurf*, being a 1D array *npmax* long. All of the y values are similarly stored in *ysurf* while the z values are store in *zsurf*. For cylindrical geometry, all of the near zone z values are stored in *zsurf*, the r values in *xsurf*. The *ysurf* array is not needed so is not created.

Reading and calculating near zone cell relevant data

For cylindrical geometry, the function **find_cell_data_rz** is called to load the near zone input geometry data into the *xsurf* and *zsurf* arrays as well as calculates the effective near zone surface and cell areas. This function first loops over the bottom and then the top of the near zone cylinder geometry. To calculate the cell area on these two surfaces, this function translates the *zsurf* node data to cell centered locations. For the n^{th} cell radius

$$\text{rcell}_n = 0.5*(xsurf_{n+1} + xsurf_n) \quad (20)$$

and the cell area is

$$cellarea_n = \pi*\text{rcell}_n*\text{rcell}_n - totalarea \quad (21)$$

where *totalarea* is the combined area of cells 1 through n-1.

For the side of the cylinder the cell radius is given from the x (or r) value of *xsurf* corresponding to the near zone cylinder side data so that for the n^{th} cell on the side surface of the near zone cylinder

$$cellarea_n = 2*\pi*\text{rcell}*\text{zcell}_n - totalarea \quad (22)$$

where

$$\text{zcell}_n = 0.5*(zsurf_{n+1} + zsurf_n) - zmin \quad (23)$$

the cell centered z location of the nth cell translated measured from the origin and *totalarea* is the combined area from cells 1 through n-1 and *zmin* is the minimum value.

Since the top and bottom of the near zone cylinder has the area π*radius*radius while the near zone cylinder side has the area 2*π*radius*(*zmax-zmin*) with *zmax* the maximum z value, these values are compared to the total area summed from the individual cells to ensure that there is agreement. These values are printed to the screen for the user to view.

For Cartesian geometry, the function **find_cell_data_xyz** is called to load the near zone geometry data into the *xsurf*, *ysurf*, and *zsurf* arrays and calculates the near zone surface areas and cell areas.

First, the function **find_surf** finds the locations of the six sides of the near zone geometry data. This function also converts the locations of the z locations of the two x-y planes of the box to integers in centimeters, assuming that the input geometry data is meters. It also converts the locations of the two y locations of the x-z planes of the near zone data box to integers in centimeters as well as the z locations of the two x-y planes of the near zone geometry box. Next, it finds the center of the near zone geometry data again in integers using centimeters. Integers are used to prevent round off errors upon comparing the total area obtained by summing the individual near zone cell areas with the total area obtained using the near zone surface dimensions.

Cells falling between edge points and points adjacent to edge points have different areas than interior cells. The function **find_edge** calculates the distance between edge points while the function **find_next_point** finds the distance between the edge points and the closest interior or adjacent point.

The function **cell_area** calculates the cell areas for the cell centered near zone geometry input data, taking into account, those cells that touch a single edge and those that touch a corner. On each surface, the cells that use edge nodes all have the same area, possibly different than interior cells. On each surface, the corner cells are assumed to have the same area, although these areas differ from both interior and edge cell areas.

In addition to calculating the individual cell areas, the total surface area is calculated. The function **print_cell_data** prints these values, the numbers of interior, edge and corner points, and the number of points, and the length between interior points in both dimensions for each surface to the screen.

The functions **read_data_rz** and **read_data_xyz** read the geometry input files for cylindrical and Cartesian coordinates respectively. The data read for **read_data_rz** are pairs of data (r, z). To switch from node to cell centered data, N2F takes for the n^{th} point

$$xsurf_n = 0.5*(r_n + r_{n-1}) \tag{24}$$

and

$$zsurf_n = 0.5*(z_n + z_{n-1}) \tag{25}$$

N2F also calculates the maximum radial extent of the near zone data (*radmax*).

The function **read_data_xyz** reads the Cartesian (x, y, z) nodal data and also calculates the maximum radial value of the near zone geometry data (*radmax*).

The far zone calculation radius (r_f) has a default value of 1.0 m. If this value is smaller than *radmax*, r_f is changed to

$$r_f = 2.0 * radmax \tag{26}$$

and a message is sent to the screen. This ensures that the radius of the far zone calculation always falls outside of the near zone field.

Having defined the near zone geometry cell properties, N2F proceeds to define and fill the near zone E and H fields arrays.

For large problems (*imem* = 1) 2D dynamic arrays are allocated (6 time the number of time points) for a single near zone geometry point. Data is repeatedly read and manipulated before the next near zone point is required. Each geometry point has 3 E and 3 H field components, each component varying in time.

For small problems, the dynamic field arrays are 3D (6 times the total number of points times the number of time steps). All components of the fields (3 E and 3 H) for every near zone geometry point for

all time values are loaded into the dynamically allocated array *ehn*. Since the input data is only read once, small problems are much faster than large ones.

For either small or large problems, *timen* and *timeh* are dynamically allocated arrays that will store time data. These arrays are 1D *ndata* long and are initialized to zero. Next, for the *ncase* number of far zone calculations to be performed, the dynamically allocated arrays *xyz_f* and *spher* contain spherical coordinate translation factors for each of the far zone points. These arrays are dynamically allocated and initialized to zero. The array *xyz_f* is 3 by *ncase* while *spher* is 4 by *ncase* in length.

The total number of time points (*ndata*) is printed to the screen as a diagnostic. N2F assumes that the input near zone time data has a uniform delta time. The function **time_step** reads the first surface (either **Fx1.flt** or **Fz1.flt**) E & H near zone data file. It reads data in groups of 7 floats. The first float in a group is the time stamp, then the 3 E field components followed by the 3 H field components. The function **get_ndata**, called earlier has already determined the number of time points. The function **time_step** loads the time stamps into the time array *timeh*, assuming that all of the near zone field data has the same time stamps. Besides loading the time array, the uniform time step (*dt*) is calculated in this function

$$\Delta t = timeh[1] - timeh[0] \tag{27}$$

the difference between the first two time stamps. The first time point, the delta time and the last time point are written to the screen as a diagnostic.

N2F next reads the near zone time varying E and H field data.

Reading E and H time dependent near zone field data

For small problems, the function **read_data_eh_rz** loads the E and H near zone data in cylindrical coordinates. First, temporary 2D storage arrays *eh* and *ehold* are created and zeroed. *eh* stores the E and H values of the n^{th} near zone geometry point while *ehold* stores the values from the previous near zone point (n-1).

For the top and bottom of the near zone cylinder, the r, ϕ, and z components of the E and H fields must be converted to Cartesian x, y, and z components. In addition, the E and H components which are node based must be converted to cell centered values. Therefore for the n^{th} point either on the top or bottom of the cylinder

$$Ex_n = 0.5*(Er_n+Er_{n-1})*\cos(\phi) - 0.5*(Ephi_n+Ephi_{n-1})*\sin(\phi)$$

$$Ey_n = 0.5*(Er_n+Er_{n-1})*\sin(\phi)+0.5*(Ephi_n+Ephi_{n-1})*\cos(\phi)$$

$$Ez_n = 0 \qquad (28)$$

$$Hx_n = 0.5*(Hr_n+Hr_{n-1})*\cos(\phi) - 0.5*(H\phi_n+H\phi_{n-1})*\sin(\phi)$$

$$Hy_n = 0.5*(Hr_n+Hr_{n-1})*\sin(\phi)+0.5*(H\phi_n+H\phi_{n-1})*\cos(\phi)$$

$$Hz_n = 0$$

Next, the E and H data for the side of the near zone cylinder is read and again translated from node to cell centered values. In this case

$$Ex_n = -0.5*(E\phi_n+E\phi_{n-1})*\sin(\phi)$$

$$Ey_n = 0.5*(E\phi_n+E\phi_{n-1})*\cos(\phi)$$

$$Ez_n = 0.5*(Ez_n+Ez_{n-1}) \qquad (29)$$

$$Hx_n = -0.5*(H\phi_n+H\phi_{n-1})*\sin(\phi)$$

$$Hy_n = 0.5*(H\phi_n+H\phi_{n-1})*\cos(\phi)$$

$$Hz_n = 0.5*(Hz_n+Hz_{n-1})$$

For cylindrical data, the E and H near zone data is the same for all near zone ϕ values. Only the $\cos(\phi)$ and $\sin(\phi)$ values in the above equations change with ϕ. The above time dependent E and H values are stored in the 3D dynamic array *ehn*(component of E or H, location, time).

The temporary 2D arrays eh and *ehold* are deleted after the *ehn* array is loaded.

The near zone data manipulation is much simpler for the Cartesian geometry, using the function **read_data_eh_xyz**. In this case the data for each component for each point for each time is simply loaded into the 3D dynamically allocated array *ehn* (component of E or H, location, time). The node to cell centered translation is performed elsewhere.

After reading the near zone geometry data and creating the necessary dynamically allocated arrays that will hold the near zone field data N2F begins far zone calculations. There are two outer loops. The most outer loop loops over the far zone ϕ, calculating n_ϕ different values, converted from input degrees to radians. The inner loop loops over the far zone θ, n_θ times, with θ converted from degrees to radians. There are *ncase* = n_ϕ*n_θ number of far zone calculations and the far zone (ϕ, θ) case index variable is *icase*. First, the sphere array is loaded for each far zone evaluation point

$$spher[0][icase] = \cos(\theta)$$

$$spher[1][icase] = \sin(\theta)$$

$$spher[2][icase] = \cos(\phi) \qquad (30)$$

$$spher[3][icase] = \sin(\phi)$$

Next, the *xyz_f* array is loaded

$$xyz\_f[0][icase] = spher[1][icase]*spher[2][icase]$$

$$xyz\_f[1][icase] = spher[1][icase]*spher[3][icase] \quad (31)$$

$$xyz\_f[2][icase] = spher[0][icase]$$

Next N2F calculates the time delay from each near zone point to each far zone point. The time delay is the time it takes the EM signal generated at a given near zone point to reach the specified far zone point travelling at the speed of light assuming a vacuum. It also calculates the maximum time delay for each far zone calculation.

For cylindrical geometry, for large problems the function **find_timed_rz_mem** finds the minimum and maximum time delay for each far zone calculation. For small problems, besides finding the minimum and maximum time delay, the time delays for all the near zone coordinates are calculated and stored in the *timedelay* array using the function **find_timed_rz**.

For both large and small problems, the function **find_timed_xyz** calculates all of the time delays and the minimum and maximum time delay for each far zone calculation.

If we assume that r_f is the radius of the far zone calculation point and r is the radius of a specific near zone coordinate, then in cylindrical coordinates

$$r = r_{surf}*\cos(\phi_n)*\sin(\theta_f)*\cos(\phi_f)+r_{surf}*\sin(\phi_n)*\sin(\theta_f)*\cos(\phi_f)+z_{surf}*\cos(\theta_f) \quad (32)$$

where “n” signifies near zone and “f” far zone

and the time delay for each near zone point to each far zone point is

timedelay[near zone point, far zone point] = $(r_f - r) / c$ calculated in the function **find_timed**.

In Cartesian coordinates

$$r = xsurf*\sin(\theta_f)*\cos(\phi_f)+ysurf*\sin(\theta_f)*\cos(\phi_f)+zsurf*\cos(\theta_f) \quad (33)$$

where “f” is far zone and *xsurf, ysurf*, and *zsurf* are the x, y, and z coordinate of each near zone point

and the time delay for each near zone point to each far zone point is again

$$timedelay = (r_f - r) /c \quad (34)$$

calculated in the function **find_timed**.

tminmax contains the maximum (*tminmax*[0]) and the minimum (*tminmax*[1]) time delays. The variable *tmin* contains the minimum time delay over all the far zone points while *tmax* contains the corresponding maximum time delay. These two values are printed to the screen.

N2F next calculates the minimum time required for a near zone signal to reach the closest far zone point, assuming that the far zone radius is 1 meter (unless the near zone data cannot be contained this sphere.

For near zone data sets which are geometrically larger than 1 m, the far zone signal is translated to the 1 meter location as commonly assumed that is the 1/R multiplying term is assumed to be 1. In addition, the –R/c term in the argument for U(t) and W(t), representing a simple phase shift is neglected.

$$addtime = tmin - (r_f - 1.)/c \tag{35}$$

We next construct a time grid that will be larger than the time interval using all the near zone and far zone coordinates. The final time for this global or reference time grid is

$$ttmax = timeh_{ndata} - timeh_0 + (tmax\text{-}tmin) \tag{36}$$

where $timeh_{ndata}$ is the last time point and $timeh_0$ is the first point of the time array.

Having previously calculated the near zone data time step, *delta_time*, we create this reference time grid using *ngrid* points with

$$ngrid = ttmax/delta\_time + 11 \tag{37}$$

where N2F has arbitrarily added 11 time steps to ensure that the universal time grid extends beyond the largest near zone time data point.

Having now determined the number of time steps to be used in the far zone calculations, N2F dynamically allocates *ehg* which contains the time points and E and H field data for each far zone calculation as well as the far zone calculation arrays *uw* and *utp*. *ehg* is 7 by *ngrid*, *uw* 12 by *ngrid* and *utp* is 4 by *ngrid*.

Next the function **time_grid** is used to create the reference time grid. Assuming that the time step of the reference time grid is the same as the time step of the near zone data (Eq. 27)

$$\Delta t = delta\_time \tag{38}$$

then the reference time grid is

$$ehg[0][ia] = ia*\Delta t \tag{39}$$

where $0 \le ia \le ngrid\text{-}1$

This time array is stored as the first component of *ehg*. The number of time nodes in this reference array, as well as the first and last time points and the time step is printed to the screen.

Having created the reference time grid, N2F loops over the number of ϕ far zone points and θ far zone points. The far zone case number is printed to the screen as well as the ϕ and θ values for this case in degrees. If the create output file option was selected on the command line (*ifileout* > 0) an output file related to the far zone ϕ and θ point number is created **etXpY.txt** where 1 ≤X goes ≤ n_θ and 1 ≤Y ≤ n_ϕ. *ehg* is zeroed, except for the first component which contains the reference time grid.

Far Zone calculation – Large Problems – Cylindrical geometry (imem = 1; igeom = 1)

For large cylindrical problems, the dynamically allocated array *ehnmem*, which contains the time dependent E and H for a single near zone point, is zeroed. For large problems, N2F loops over each of the 3 near zone surfaces. The arrays *eh* and *ehold*, being 6 by *ndata*, are zeroed before reading in the near zone data for a surface. The time dependent E and H field data for the first point on a surface is read and loaded into *ehold*. After reading this data, a message is sent to the screen indicating that the first of N points on surface S has been read, where N is the total number of near zone points on surface S, going from 1 to 3.

N2F then loops over each near zone point on a surface, which for cylindrical geometry is a line (N2F replicates the near zone data over the near zone ϕ).

After every 10^{th} point is read, a progress message is sent to the screen. After reading the E and H field data for a near zone point, the function **timedelay_rz** is called to calculate the time delay at each replicated ϕ value from the near zone point to the given far zone point.

For each near zone replicated ϕ data value, N2F time shifts the input near zone time data

$$timen[\mathrm{i}] = timeh[\mathrm{i}] + timedelay - tmin \tag{40}$$

where 1 ≤ I ≤ *ndata*, *timeh* contains the original time data and *tmin* is the minimum time delay.

Next the near Zone E and H data is transformed to Cartesian coordinates and translated from node to cell centered values

$$\begin{aligned}
ehnmem[0] &= 0.5*(eh[0]+ehold[0])*\cos(\phi) - 0.5*(eh[1]+ehold[1])*\sin(\phi) \\
ehnmem[1] &= 0.5*(eh[0]+ehold[0])*\sin(\phi) + 0.5*(eh[1]+ehold[1])*\cos(\phi) \\
ehnmem[2] &= 0.0 \\
ehnmem[3] &= 0.5*(eh[3]+ehold[3])*\cos(\phi) - 0.5*(eh[4]+ehold[4])*\sin(\phi) \\
ehnmem[4] &= 0.5*(eh[3]+ehold[3])*\sin(\phi) + 0.5*(eh[4]+ehold[4])*\cos(\phi)
\end{aligned} \tag{41}$$

$$ehnmem[5] = 0.0$$

Having switched to cell centered values with the appropriate time shift, N2F interpolates these time dependent values onto the reference time grid using the function **interpmem** loading the dynamically allocated array *ehg* with the interpolated values.

The function **interpmem** loops over the reference time grid. If the reference time grid points are less than the starting time point of the time shifted near zone data, then these time grid data points are assigned the values of the near zone E and H minimum time values, which are typically zero.

If the reference time grid points are greater than the maximum time shifted time of the near zone point, then these reference time grid points are assigned the value of E and H from the maximum time shifted value for E and H, again typically zero.

For all the reference time grid points between these two extremes, a linear interpolation in time is used to translate from the near zone data to the reference grid. If for some reason the interpolation is not successful, an error message is printed to the screen, an error file named **error.txt** is created and the "offending" data point written to this file; then N2F terminates.

Having successfully interpolated the near zone cell centered E and H data to the reference time grid; the function **int_uw_ends** performs the appropriate surface cell integrations on the bottom and top of the near zone cylinder.

For each time step, for each near zone coordinate for each replicated ϕ value, the cell centered time shifted onto the reference grid E and H data (stored in the dynamically allocated *ehg*) is transformed from Cartesian to Cylindrical coordinates

$$\begin{aligned}
temprpz[0] &= ehg[1]*\cos(\phi)-ehg[2]*\sin(\phi) \\
temprpz[1] &= -ehg[1]*\sin(\phi)+ehg[2]*\cos(\phi) \\
temprpz[2] &= ehg[3] \\
temprpz[3] &= ehg[4]*\cos(\phi)-ehg[5]*\sin(\phi) \\
temprpz[4] &= -ehg[4]*\sin(\phi)+ehg[5]*\cos(\phi) \\
temprpz[5] &= ehg[6]
\end{aligned} \tag{42}$$

with *ehg*[0] containing the reference time grid. The function then calculates the cross products E × n and H × n where n is the surface normal vector to the bottom and top of the near zone cylinder, disregarding the z component of the cross products which is zero

$$\begin{aligned}
crossrpz[0] &= temprpz[1] \\
crossrpz[1] &= -temprpz[0]
\end{aligned}$$

$$crossrpz[2] = 0.0$$

$$crossrpz[3] = temprpz[4] \tag{43}$$

$$crossrpz[4] = temprpz[3]$$

$$crossrpz[5] = 0.0$$

Next, the function transforms back to Cartesian coordinates from the cylindrical coordinates, integrates over each cell area, and sums assuming that the cell centered E and H values are constant over a cell area

$$uw[0] = \Sigma\ (\ norm[is]*(crossrpz[0]*\cos(\phi)-crossrpz[1]*\sin(\phi))*cellarea/cnst)$$

$$uw[1] = \Sigma\ (\ norm[is]*(crossrpz[0]*\sin(\phi)+crossrpz[1]*\cos(\phi))*cellarea/cnst)$$

$$uw[3] = \Sigma(-\ norm[is]*(crossrpz[3]*\cos(\phi)-crossrpz[4]*\sin(\phi))*cellarea/cnst) \tag{44}$$

$$uw[4] = \Sigma\ (-\ norm[is]*(crossrpz[3]*\sin(\phi)+crossrpz[4]*\cos(\phi)*cellarea/cnst)$$

where $norm[is]$ = -1 for the bottom of the cylinder (is = 0) and 1 for the top (is=1) and $cnst$ is the normalizing constant $4 * \pi * c$ (with R set to 1 in Eqs. 4-5).

After the uw values are calculated for a given near zone point (including the replicated ϕ values) the E and H data from the original near zone data is loaded into the $ehold$ array and the next near zone point data is read into the eh array so that the next cell centered interpolated point can be calculated.

After all of the bottom and top cells have been integrated, a similar procedure is used to add the cylinder surface (is = 2) near zone data, again replicated $nphi\_data$ times to the far zone calculation.

Again, the near zone cylinder side data file is opened and the E and H data loaded into $ehold$. The next point near zone time data is loaded into $timeh$ and the E and H data loaded into eh. For each ϕ value the $timedelay$ is calculated as before and a time shifted new time array associated with each of the replicated near zone points is loaded into $timen$. The near zone data is then translated from node to cell centered values

$$ehnmem[0] = -0.5*(eh[1] +ehold[1])*\sin(\phi)$$

$$ehnmem[1] = 0.5*(eh[1]+ehold[1])*\cos(\phi)$$

$$ehnmem[2] = 0.5*(eh[2]+ehold[2])$$

$$ehnmem[3] = -0.5*(eh[4]+ehold[4])*\sin(\phi) \tag{45}$$

$$ehnmem[4] = 0.5*(eh[4]+ehold[4])*\cos(\phi)$$

$$ehnmem[5] = 0.5*(eh[5]+ehold[5])$$

Having switched to cell centered values with the appropriate time shift, N2F interpolates these time dependent values onto the reference time grid using the function **interpmem** loading the dynamically allocated array *ehg* with the interpolated values.

Having successfully interpolated the near zone cell centered E and H data to the reference time grid, the function **int_uw_side** performs the appropriate surface cell integrations on side of the near zone cylinder

For each time step, for each near zone coordinate for each replicated ϕ value, the cell centered time shifted onto the reference grid E and H data (stored in the *ehg* dynamically allocated array) is first transformed from Cartesian to Cylindrical coordinates

$$\begin{aligned}
temprpz[0] &= ehg[1]*\cos(\phi)-ehg[2]*\sin(\phi) \\
temprpz[1] &= -ehg[1]*\sin(\phi)+ehg[2]*\cos(\phi) \\
temprpz[2] &= ehg[3] \\
temprpz[3] &= ehg[4]*\cos(\phi)-ehg[5]*\sin(\phi) \\
temprpz[4] &= -ehg[4]*\sin(\phi)+ehg[5]*\cos(\phi) \\
temprpz[5] &= ehg[6]
\end{aligned} \tag{46}$$

remembering that *ehg*[0] contains the reference time grid. The function nexts calculates the cross products E × n and H × n where n is the surface normal vector to side of the near zone cylinder and disregarding the z component of these cross product which is zero

$$\begin{aligned}
crossrpz[0] &= 0.0 \\
crossrpz[1] &= temprpz[2] \\
crossrpz[2] &= -temprpz[1] \\
crossrpz[3] &= 0.0 \\
crossrpz[4] &= temprpz[5] \\
crossrpz[5] &= -temprpz[4]
\end{aligned} \tag{47}$$

Next, the function transforms back to Cartesian coordinates from the cylindrical coordinates and integrates over each cell area and sums, assuming that the cell centered E and H values are constant over a cell area

$$uw[0] = \Sigma ((crossrpz[0]*\cos(\phi)-crossrpz[1]*\sin(\phi))*cellarea/cnst)$$

$$uw[1] = \Sigma ((crossrpz[0]*\sin(\phi)+crossrpz[1]*\cos(\phi))*cellarea/cnst)$$

$$uw[2] = \Sigma(crossrpz[2]*cellarea/cnst)$$

$$uw[3] = -\Sigma ((crossrpz[3]*\cos(\phi)-crossrpz[4]*\sin(\phi))*cellarea/cnst) \quad (48)$$

$$uw[4] = \Sigma ((crossrpz[3]*\sin(\phi)+crossrpz[4]*\cos(\phi))*cellarea/cnst)$$

$$uw[5] = -\Sigma(crossrpz[2]*cellarea/cnst)$$

where *cnst* is the normalizing constant $4 * \pi * c$ (with R in Eqs 4-5 set to 1)

After the *uw* values are calculated for a given near zone point (including the replicated ϕ values) the E and H data from the original near zone data is loaded into *ehold* and the next near zone data is read into *eh* so that the next cell centered interpolated point can be calculated.

If *igeom* is not 1, then the geometry is Cartesian and transformations involving cylindrical coordinates are not required.

Far Zone calculation – Large Problems – Cartesian geometry (*imem* = 1; *igeom* = 0)

N2F loops over each near zone point on each surface of the near zone block. After every 10^{th} point is read, a progress message is sent to the screen. After reading the E and H field data for a near zone point, the time delay previously calculated for each near zone point is added to the time array for each near zone point. For Cartesian geometries, the *ehnmem* array simply contains the E and H data, no coordinate transformation is required,

The interpolation function **interpmem** is called, which interpolates the E and H near zone time data onto the reference time grid. Then the integration routine function **int_uw** integrates E × n and H × n over each cell on the near zone block.

Function **int_uw** is straightforward in Cartesian coordinates using the previously calculated near zone cell areas.

First, the –x and +x surface data is used

$$uw[1] = \Sigma (norm*ehg[3]*cellarea/cnst)$$

$$uw[2] = \Sigma (-norm*ehg[2]*cellarea/cnst)$$

$$uw[4] = \Sigma(norm*ehg[6]*cellarea/cnst) \quad (49)$$

$$uw[5] = \Sigma(-norm*ehg[5]*cellarea/cnst)$$

where *norm* is -1 for –x and 1 for +x surface

Next, the –y and +y surface data is used

$$uw[1] = \Sigma(-norm*ehg[3]*cellarea/cnst)$$

$$uw[2] = \Sigma(norm*ehg[1]*cellarea/cnst)$$

$$uw[3] = \Sigma(-norm*ehg[6]*cellarea/cnst) \quad (50)$$

$$uw[5] = \Sigma(norm*ehg[6]*cellarea/cnst)$$

where *norm* is -1 for –y and 1 for +y surface

Finally, the –z and +z surface data is used

$$uw[0] = \Sigma(norm*ehg[2]*cellarea/cnst)$$

$$uw[1] = \Sigma(-norm*ehg[1]*cellarea/cnst)$$

$$uw[3] = \Sigma(-norm*ehg[5]*cellarea/cnst) \quad (51)$$

$$uw[4] = \Sigma(norm*ehg[4]*cellarea/cnst)$$

where *norm* is -1 for –z and 1 for +z surface

and *cnst* is the constant $4*\pi * c$ (with R = 1 found in Eqs 4-5)

Far Zone calculation – Small Problems – Cylindrical geometry (*imem* = 0; *igeom* = 1)

For small problems, all of the time varying E and H data for all the near zone points fit into memory and have been previously loaded into the dynamically allocated *ehn* array. First the bottom of the near zone cylinder data points are used, then the top of the cylinder. Finally the cylinder side data is used. For each near zone point on each of these surfaces, for each of the replicated *nphi_data* angles, the previously calculated time delay is added to the time array. The function **interp** then interpolates the near zone input time onto the reference time grid.

Function **interp** is similar to **interpmem** except that unlike i**nterpmem**, the time delay array contains all of the time delay data, based on the distance from each replicated near zone point. This function loops over the reference time grid. If the reference time grid points are less than the starting time point of the time shifted near zone data, then these time grid data points are assigned the values of the near zone E and H minimum time values, which are typically zero.

If the reference time grid points are greater than the maximum time shifted time of the near zone point, then these reference time grid points are assigned the value of E and H from the maximum time shifted value for E and H, again typically zero.

For all the reference time grid points between these two extremes, a linear interpolation in time is used to translate from the near zone data to the reference grid. If for some reason the interpolation is not successful, an error message is printed to the screen, an error file named **error.txt** is created and the "offending" data point written to this file, then N2F terminates.

Having successfully interpolated the near zone cell centered E and H data to the reference time grid; the function **int_uw_ends** performs the appropriate surface cell integrations on the bottom and top of the near zone cylinder.

For each time step, for each near zone coordinate for each replicated ϕ value, the cell centered time shifted onto the reference grid E and H data (stored in dynamically allocated *ehg*) is transformed from Cartesian to Cylindrical coordinates

$$\begin{aligned}
temprpz[0] &= ehg[1]*\cos(\phi)-ehg[2]*\sin(\phi) \\
temprpz[1] &= -ehg[1]*\sin(\phi)+ehg[2]*\cos(\phi) \\
temprpz[2] &= ehg[3] \\
temprpz[3] &= ehg[4]*\cos(\phi)-ehg[5]*\sin(\phi) \\
temprpz[4] &= -ehg[4]*\sin(\phi)+ehg[5]*\cos(\phi) \\
temprpz[5] &= ehg[6]
\end{aligned} \tag{52}$$

with *ehg*[0] containing the reference time grid. The function next calculates the cross products E ×n and H × n where n is the surface normal vector to the bottom and top of the near zone cylinder and disregarding the z component of the cross products which is zero

$$\begin{aligned}
crossrpz[0] &= temprpz[1] \\
crossrpz[1] &= -temprpz[0] \\
crossrpz[2] &= 0.0 \\
crossrpz[3] &= temprpz[4] \\
crossrpz[4] &= temprpz[3] \\
crossrpz[5] &= 0.0
\end{aligned} \tag{53}$$

Next, the function transforms back to Cartesian coordinates from the cylindrical coordinates and integrates over each cell area and sums, assuming that the cell centered E and H values are constant over a cell area

$$uw[0] = \Sigma\ (\ norm[is]*(crossrpz[0]*\cos(\phi)-crossrpz[1]*\sin(\phi))*cellarea/cnst)$$

$$uw[1] = \Sigma\ (\ norm[is]*(crossrpz[0]*\sin(\phi)+crossrpz[1]*\cos(\phi))*cellarea/cnst)$$

$$uw[3] = \Sigma\ (-\ norm[is]*(crossrpz[3]*\cos(\phi)-crossrpz[4]*\sin(\phi))*cellarea/cnst) \quad (54)$$

$$uw[4] = \Sigma\ (-norm[is]*(crossrpz[3]*\sin(\phi)+crossrpz[4]*\cos(\phi))*cellarea/cnst)$$

where *norm*[*is*] = -1 for the bottom of the cylinder *(is* = 0) and 1 for the top (*is*=1) and *cnst* is the normalizing constant 4 * π * c (with R = 1 found in Eqs.4-5)

After all of the bottom and top cells have been integrated, a similar procedure is used to add the cylinder surface (is = 2) near zone data, again replicated *nphi_data* times for the far zone calculation.

The near zone time data is loaded into *timeh* and the E and H data loaded into *eh*. For each ϕ value the time delay is calculated as before and a time shifted new time array associated with each of the replicated near zone points is loaded into *timen*. The near zone data is then translated from node to cell centered values

$$ehnmem[0] = -0.5*(eh[1] + ehold[1])*\sin(\phi)$$

$$ehnmem[1] = 0.5*(eh[1]+ehold[1])*\cos(\phi)$$

$$ehnmem[2] = 0.5*(eh[2]+ehold[2])$$

$$ehnmem[3] = -0.5*(eh[4]+ehold[4])*\sin(\phi) \quad (55)$$

$$ehnmem[4] = 0.5*(eh[4]+ehold[4])*\cos(\phi)$$

$$ehnmem[5] = 0.5*(eh[5]+ehold[5])$$

Having switched to cell centered values with the appropriate time shift, N2F interpolates these time dependent values onto the reference time grid using the function **interp** loading the dynamically allocated *ehg* with the interpolated values.

Having successfully interpolated the near zone cell centered E and H data to the reference time grid, the function **int_uw_side** performs the appropriate surface cell integrations on side of the near zone cylinder

For each time step, for each near zone coordinate for each replicated ϕ value, the cell centered time shifted onto the reference grid E and H data (stored in dynamically allocated *ehg*) is first transformed from Cartesian to cylindrical coordinates

$$temprpz[0] = ehg[1]*\cos(\phi)-ehg[2]*\sin(\phi)$$

$$temprpz[1] = -ehg[1]*\sin(\phi)+ehg[2]*\cos(\phi)$$

$$temprpz[2] = ehg[3]$$

$$temprpz[3] = ehg[4]*\cos(\phi)-ehg[5]*\sin(\phi) \tag{56}$$

$$temprpz[4] = -ehg[4]*\sin(\phi)+ehg[5]*\cos(\phi)$$

$$temprpz[5] = ehg[6]$$

remembering that *ehg[*0] contains the reference time grid. The function then calculates the cross products E × n and H × n where n is the surface normal vector to side of the near zone cylinder and disregarding the z component of the cross products which is zero

$$crossrpz[0] = 0.0$$

$$crossrpz[1] = temprpz[2]$$

$$crossrpz[2] = -temprpz[1]$$

$$crossrpz[3] = 0.0 \tag{57}$$

$$crossrpz[4] = temprpz[5]$$

$$crossrpz[5] = -temprpz[4]$$

Next, the function transforms back to Cartesian coordinates from the cylindrical coordinates and integrates over each cell area and sums, assuming that the cell centered E and H values are constant over a cell area

$$uw[0] = \Sigma ((crossrpz[0]*\cos(\phi)-crossrpz[1]*\sin(\phi))*cellarea/cnst)$$

$$uw[1] = \Sigma ((crossrpz[0]*\sin(\phi)+crossrpz[1]*\cos(\phi))*cellarea/cnst)$$

$$uw[2] = \Sigma(crossrpz[2]*cellarea/cnst)$$

$$uw[3] = -\Sigma ((crossrpz[3]*\cos(\phi)-crossrpz[4]*\sin(\phi))*cellarea/cnst) \tag{58}$$

$$uw[4] = \Sigma ((crossrpz[3]*\sin(\phi)+crossrpz[4]*\cos(\phi))*cellarea/cnst)$$

$$uw[5] = -\Sigma(crossrpz[2]*cellarea/cnst)$$

where *cnst* is the normalizing constant $4 * \pi * c$ (where R = 1 found in Eqs. 4-5)

Far Zone calculation – Small Problems – Cartesian geometry (*imem* = 0; *igeom* = 0)

N2F loops over each near zone point on each surface of the near zone block. Being a small problem all of the near zone E and H field data has been previously loaded into *ehn*. The time delay previously calculated for each near zone point is added to the time array for each near zone point. For Cartesian geometries, *ehn* contains the E and H data, no coordinate transformation is required.

Next, the interpolation function **interp** is called, which interpolates the E and H near zone time data onto the reference time grid. Then the integration function **int_uw** integrates E × n and H × n over each cell on the near zone block surfaces.

Function **int_uw** is straightforward in Cartesian coordinates using the previously calculated near zone cell areas.

First, the –x and +x surface data is used

$$\begin{aligned} uw[1] &= \Sigma(norm * ehg[3] * cellarea/cnst) \\ uw[2] &= \Sigma(-norm * ehg[2] * cellarea/cnst) \\ uw[4] &= \Sigma(norm * ehg[6] * cellarea/cnst) \\ uw[5] &= \Sigma(-norm * ehg[5] * cellarea/cnst) \end{aligned} \tag{59}$$

where *norm* is -1 for –x and 1 for +x surface

Next, the –y and +y surface data is used

$$\begin{aligned} uw[1] &= \Sigma(-norm * ehg[3] * cellarea/cnst) \\ uw[2] &= \Sigma(norm * ehg[1] * cellarea/cnst) \\ uw[3] &= \Sigma(-norm * ehg[6] * cellarea/cnst) \\ uw[5] &= \Sigma(norm * ehg[6] * cellarea/cnst) \end{aligned} \tag{60}$$

where *norm* is -1 for –y and 1 for +y surface

Finally, the –z and +z surface data is used

$$\begin{aligned} uw[0] &= \Sigma(norm * ehg[2] * cellarea/cnst) \\ uw[1] &= \Sigma(-norm * ehg[1] * cellarea/cnst) \\ uw[3] &= \Sigma(-norm * ehg[5] * cellarea/cnst) \end{aligned} \tag{61}$$

$$uw[4] = \Sigma(norm*ehg[4]*cellarea/cnst)$$

where *norm* is -1 for –x and 1 for +x surface and

cnst is the constant $4*\pi$ * c where R = 1 found in Eqs.4-5

Calculate derivatives of u and w for both large and problems, Cartesian and Cylindrical coordinates

After calculating the integrated u and w in Cartesian coordinates (using transformed arrays from cylindrical geometry if required) N2F calls **derive_uw** to calculate the time derivative of the u and w. This function calculates the time derivatives of the 6 components of the *uw* array. The first three components of *uw* contain the u values (u_x, u_y, u_z) , the next 3 contain the w values (w_x, w_y, w_z)

The 6 time derivative vector values are loaded into the last 6 locations of *uw* (*uw* being 12 x *ngrid*, where *ngrid* is the number of reference time points). *uw* contain the values for the following 12 variables

u_x u_y u_z w_x w_y w_z $\frac{\partial u_x}{\partial t}$ $\frac{\partial u_y}{\partial t}$ $\frac{\partial u_z}{\partial t}$ $\frac{\partial w_x}{\partial t}$ $\frac{\partial w_y}{\partial t}$ $\frac{\partial w_z}{\partial t}$

For the first time grid point

$$uw[i+6][0] = (uw[i][1] - uw[i][0]) / \Delta t \quad (62)$$

$1 \le i \le 6$ and Δt is the reference time grid step size. This is a one-sided time derivative and is usually zero, assuming that the reference time grid encompasses all of the near zone time data, shifted to the far zone point. For all the time points between these extremes, a two sided time derivative is used

$$uw[6+i][ia] = (uw[i][ia+1] - uw[i][ia-1])/(2*\Delta t) \quad (63)$$

where *ia* goes from the second time point to the next to last time point (*ngrid*-1) and $1 \le i \le 6$).

The final time grid derivative is set equal to the derivative of the previous time point.

$$uw[i+6][ngrid] = uw[i][ngrid-1] \quad (64)$$

Since the reference time grid should encompass all of the time values, these derivatives are usually zero.

Finally, N2F transforms these derivatives from Cartesian to Spherical coordinates using the function **transform_uw**. This function transforms the du/dt and dw/dt values from Cartesian to spherical coordinates without calculating the unrequired r component.

utp[0][*ia*] contains the θ component of du/dt for the *ia* time grid point

utp[1][*ia*] contains the ϕ component of du/dt for the *ia* time grid point

utp[2][*ia*] contains the θ component of dw/dt for the *ia* time grid point

utp[3][*ia*] contains the ϕ component of dw/dt for the *ia* time grid point

Final Calculations

Looping over all the time grid points $1 \le ia \le ngrid$ Using Eqs 1-2, N2F calculates

$$E_\theta = -(\eta * utp[2][ia] + utp[1][ia]) \quad (65)$$

and

$$E_\phi = -\eta * utp[3][ia] + utp[0][ia] \quad (66)$$

N2F prints the maximum absolute value of E_θ and E_ϕ and the time where these values occur, assuming that these values occur at a time t

$$t > tmax + addtime \quad (67)$$

where *tmax* is the maximum time delay and *addtime* is from Eq. 35 to prevent N2F from finding spurious signal spikes from times earlier than the time it takes a signal to arrive at the far zone point from the farthest near zone point. For these earlier times, the calculation is not valid.

N2F prints to the summary file (sum.txt) and the screen for each far zone θ and ϕ value

i_θ i_ϕ θ ϕ t_θ E_θ t_ϕ E_ϕ where

i_θ is the index of the loop over θ corresponding to the maximum absolute value of E_θ

i_ϕ is the index of the loop over ϕ corresponding to the maximum absolute value of E_ϕ

θ is the angle in degrees corresponding i_θ

ϕ is the angle in degrees corresponding i_ϕ

t_θ is the time where E_θ is maximum of the absolute value

t_ϕ is the time where E_ϕ is maximum of the absolute value

E_θ is the maximum of the absolute value

E_ϕ is the maximum of the absolute value

If the output flag *ifileout* > 0, then for each far zone θ and ϕ value, an output file is created, each file having three columns of data: time t, $E_\theta(t)$, and $E_\phi(t)$.

After writing each far zone point data file and the summary file, N2F execution ends.

List of Functions

void **itoa**(int n, char *s)

converts integer n > 0 to character s. Used to create name of output file.

void **reverse**(char *s)

reverses order of characters in s. Used to create name of output file.

void **interp**(int ic, double **timen*, double ****ehn,* int *ndata*, double ***ehg,* int *ngrid*, double *delta_time*)

For small problems, for far zone point index ic, using the reference time grid *timen* which contains *ngrid* time points, this function linearly interpolates the *ndata* time points of the near zone E and H data contained in *ehn* onto the reference time grid and stores the new results in *ehg*. The linear interpolation uses the time grid Δt, *delta_time*.

void **interpmem**(double **timen*, double ****ehn*, int *ndata*, double ***ehg*, int *ngrid*, double *delta_time*)

For large problems, using the reference time grid *timen* which contains *ngrid* time points, this function linearly interpolates the *ndata* time points of the near zone E and H data contained in e*hn* onto the reference time grid and stores the new results in *ehg*. The linear interpolation uses the time grid Δt, *delta_time*.

int **alloc1D**(double* &ptr, int M)

creates a 1 dimensional double array of length M returning 1 if successful.

int **alloc2D**(double** &ptr, int M, int N)

creates a 2 dimensional double array M x N returning 1 if successful.

int **alloc3D**(double** &ptr, int L, int M, int N)

creates a 3 dimensional double array L x M x N returning 1 if successful.

int **zero1D**(double* &ptr, int M)

zeros a 1 dimensional double array of length M returning 1 if successful.

int **zero2D**(double** &ptr, int M, int N)

zeros a 2 dimensional double array M x N returning 1 if successful.

int **zero3D**(double** &ptr, int L, int M, int N)

zeros a 3 dimensional double array L x M x N returning 1 if successful.

double **time_step**(int *ndata*, double **timeh*)

opens a near zone point input field file, reads *ndata* sets of 7 floats (1 time, 3 E field and 3 H field components) creates the near zone time *timeh* which assumes that all the near zone point data uses the same time grid and assuming a uniform time grid, return the time step Δt, the difference between the second and first input time points.

void **time_grid**(int *ngrid*, double *dt*, double ***ehg*)

creates the reference time grid containing *ngrid* time points with a uniform time step of *dt* and places this in the global E and H array in the first location *ehg*[0][*ia*] where 0 ≤ *ia* ≤ *ngrid*-1. First reference time point is 0.

void **int_uw**(int *ngrid*, int is, int ii, double **cellarea*, double ***ehg*, double ***uw*, double *cnst*)

calculates for Cartesian problems, the integral in Eqs. 4-5 not including the time derivative. The number of reference time grid points *ngrid*. The near zone point used in the integration is ii on the near zone surface is. The cell area is *cellarea*, the field data is stored in the *ehg*, the integrated value stored in *uw* and *cnst* is the constant value outside of the integral in Eqs. 4-5 with R set to 1.

void **deriv_uw**(int *ngrid*, double *delta_t*, double ***uw*)

performs the time derivative found in Eqs. 4-5 on the first 6 columns of the *uw* , storing the results in the last 6 columns. The time array has *ngrid* time points with uniform time step *delta_t*. A central difference numerical derivative is used.

void **zero_ref_time_arrays**(int *ngrid*, double ***ehg*, double ***uw*, double ***utp*)

zeros the dynamically created *ehg*, *uw* and *utp*. While *ehg* is 7 by *ngrid*, the first column contains the reference time grid so this column is not zeroed. *uw* grid is 12 x *ngrid* and *utp* is 4 x ngrid.

int **get_npoints**(int *igeom*, int *is*)

returns the number of near zone data points in a near zone geometry file for surface *is*, assuming 4 bytes per float. For a cylindrical geometry, there are two floats per near zone point (r, z) while for Cartesian there are 3 floats (x, y, z) per point.

int **get_ndata**(int *igeom*, int n)

returns the number of time points associated with the near zone field data, assuming that all the near zone points have the same number of time points. The function opens **Fx1.flt** containing n near zone points for Cartesian geometries. For cylindrical coordinates, adds 1 to the n number of near zone points contained in the field field **Fz1.flt** that is read. N2F assumes 4 bytes per float and that the field file contains n (or n+1) sets of 7 groups of floats (1 time, 3 E field, and 3 H field components) after counting the number of bytes in the file.

void **transform_uw**(int *icase*, int *ngrid*, double **uw*, double **utp*, double **spher*)

transforms the results of Eqs. 4-5 from Cartesian to spherical coordinates. Loops over the *ngrid* reference time data contained in the last 6 columns of *uw* transforming from Cartesian to spherical coordinates using the transformations stores in *spher* for each *icase* far zone point. The result of this transformation is stored in *utp* (4 x *ngrid*).

double **read_data_xyz**(int **npoints*, double **xsurf*, double **ysurf*, double **zsurf*)

For each near zone Cartesian geometry input file, reads the *npoints* number of near zone geometry data points, loading the x component values into *xsurf*, the y component values into *yzsurf*, and the z component values into the *zsurf*. Returns distance from the origin to near zone point furthest from the origin, assuming that the near zone surface surrounds the origin. If this is not the case, then the near zone data should be translated to surround the origin before using N2F.

double **read_data_rz**(int **npoints*, double **xsurf*, double **zsurf*)

For each near zone cylindrical geometry input file, reads the *npoints* number of near zone geometry data points, loading the r component values into *xsurf*, and the z component values into *zsurf* after transforming from node to cell centered values. Returns distance from the origin to near zone point furthest from the origin, assuming that the near zone surface surrounds the origin. If this is not the case, then the near zone data should be translated to surround the origin before using N2F.

void **find_surf**(int **npend*, int **lsurf*, double **xsurf*, double **ysurf*, double **zsurf*)

For Cartesian problems, N2F assumes that the input geometry data is in meters. After summing the number of near zone geometry points over all 6 near zone surfaces, *npend* contains the index of the first near zone point on each of the 6 near zone surfaces. This functions loads into *lsurf*, the x locations of the two near zone y-z planes, the y locations of the two x-z planes and the z locations of the two x-y planes in centimeters using integers. N2F assumes that the near zone surfaces all lie on integer centimeter values.

void **find_timed_rz**(int *icase* int **npoints*, double ***xyz_f*, double* *tminmax*, double *r_f*, double **xsurf*, double **zsurf*, int *nph*i, double **cosp*, double **sinp*, double ***timedelay*)

finds the time delay = $(\bar{r}'.\hat{r})$/c for small problems using cylindrical coordinates, including the replicated values, for the *icase* far zone point located at the *xyz_f* Cartesian coordinates. Besides loading the time delay value into *timedelay*, this also finds the minimum time delay, *tminmax*[0], and the maximum time delay, *tminmax*[1]. *cosp* and *sinp* contain *nphi* geometric factors used to transform the ϕ replicated values from cylindrical to Cartesian coordinates.

void **find_timed_rz_mem**(int *icase* int **npoints*, double ***xyz_f*, double* *tminmax*, double *r_f*, double **xsurf*, double **zsurf*, int *nphi*, double **cosp*, double **sinp*)

finds the minimum and maximum time delay = $(\bar{r}'.\hat{r})$/c for large problems using cylindrical coordinates, including the replicated values, for the *icase* far zone point located at the *xyz_f* Cartesian coordinates. The minimum time delay found is *tminmax*[0], and the maximum time delay found, *tminmax*[1]. *cosp* and *sinp* contain *nphi* geometric factors used to transform the ϕ replicated values from cylindrical to Cartesian coordinates.

double **find_timed**(double *rpime, double *rhat)

returns the time delay R/c $-(\bar{r}'.\hat{r})$/c used in Eqs. 4-5 where R is the radius of the far zone point. This is typically 1 meter, or twice the distance from the origin to the farthest near zone point, if this distance is greater than 1 meter.

double **timedelay_rz**(int *icase* int *ip*, int *icp*, double **xyz_f*, double* *tminmax*, double *r_f*, double *xsurf, double *zsurf, double *cosp, double *sinp)

for large cylindrical problems, returns the time delay for the *icase* far zone point, for the *ip* ϕ replicated angle, for the *icp* near zone point, using the function **find_timed** and the cylindrical to Cartesian transformations *cosp* and *sinp*.

void **find_timed_xyz**(int *icase* int *npoints, double **xyz_f*, double* *tminmax*, double *r_f*, double *xsurf, double *ysurf, double *zsurf, double **timedelay*)

for small Cartesian problems, for the *icase* far zone point finds the time delay = $(\bar{r}'.\hat{r})$/c located at the *xyz_f* Cartesian coordinates. Besides loading the time delay value into *timedelay*, this also finds the minimum time delay, *tminmax*[0], and the maximum time delay, *tminmax*[1].

void **find_edge**(int *npoints, double *xsurf, double *ysurf, double *zsurf)

for Cartesian problems, finds the location of all the near zone surface edges and corners points in centimeters, assuming that these values should be integers, checking all the *npoints* on each near zone surface.

void **find_next_point**(int *npoints, double *xsurf, double *ysurf, double *zsurf)

for Cartesian problems, finds on each of the 6 near zone surfaces the distance between the minimum and maximum points in each of the two dimensions to the adjacent point in centimeters, assuming that these values should be integers, checking all the *npoints* on each near zone surface. These values are used to calculate the cell areas that have surface edges and/or corners as boundaries.

void **cell_area**(int *npoints, int *lsurf, double *surfacearea, double *cellarea, double *xsurf, double *ysurf, double *zsurf)

for Cartesian problems, calculates the near zone cells areas, looping over all of the *npoints* cells on each near zone surface. Also calculates the surface area of each of the 6 near zone surfaces. Uses integer arithmetic to avoid roundoff errors, assuming that the surfaces contain an integer number of square centimeters. Takes into account the possibility that corner and edge cell areas may have areas different

than the surface interior cells. Each of the 6 near zone surfaces can have differing, but uniform grid spacing. *lsurf* contains the location (the third dimension) in integers of each of the 6 planes.

double **find_cell_data_rz**(int *_npoints_, double *_xsurf_, double *_zsurf_, double *_cellarea_)

for cylindrical problems, calculates and loads into *cellarea*, the cells areas of the near zone points. Also calculates the total surface area of each of the three (bottom, top, and side) of the near zone data cylinder summing over the *npoints* cell areas on each of the surfaces. Checks the surface area of each of the three sides against the surface area using the cylinder dimensions. The locations of the near zone points are in *xsurf* and *zsurf* . This function returns the distance from the origin to the farthest near zone point.

double **find_cell_data_xyz**(int *_lsurf_, int *_npoints_, double *_xsurf_, double *_ysurf_, double *_zsurf_, double *_surfacearea_, double *_cellarea_)

for Cartesian problems, using the six values associated with the third dimension in *lsurf* for the location of the 6 planes converted to integer centimeters that comprise the near zone geometry data stored in *xsurf*, *ysurf*, and *zsurf*, this function returns the distance from the origin to the farthest near zone point. This function also passes up the cell areas stored in *cellarea* and the surface areas in *surfacearea*.

void **print_cell_data**(int *_int_pnts_, int *_edge_pnts_, int *_corner_pnts_, double *_surfacearea_, int *_lsurf_)

for Cartesian problems, for each of the six near zone surfaces, prints the number of interior points, edge points, corner points and the sum . This also prints the location of each of the six near zone planes contained in *lsurf* assuming these lie on integer number of centimeters, the surface areas of these planes, contained in *surfacearea*, the surface areas for these planes calculated from the dimensions of the planes, and the length of each of the two dimensions for each plane. These should be used to check that the Cartesian geometry data was correctly read.

int **ifound**(int *ic*, int *ia*, int *ndata*, double *delta_time*, double** *ehg*, double *_timen_, double*** *ehn*, int *loc*)

for small problems, for the *ic* near zone point, this returns the location in the input time grid which contains the reference time *ehg*[0][*ia*] for a given *ia*. The input time grid is *timen* and this routine returns I when *timen*[i] ≤ *ehg*[0][*ia*] < *timen*[i+1]. This function also linearly interpolates the E and H near zone data contained in *ehn* onto the *ehg*[0] reference grid at the *ia* grid point and loads these values into the next 6 columns of *ehg* . This routine loops from the *loc* input time point to the *ndata*-1 time point. Since both time grids, the input and the reference are monotonically increasing keeping track of the location of the previously found time point returned by the previous call, stored in *loc*, this routine is faster, than starting at the first location. The time step *delta_time* is used in the linear interpolation.

int **ifoundmem**(int *ndata*, double *delta_time*, double** *ehg*, double *_timen_, double*** *ehn*, int *loc*)

for large problems this returns the location in the input time grid which contains the reference time *ehg*[0][*ia*] for a given *ia*. The input time grid is *timen* and this routine returns I when *timen*[i] ≤ *ehg*[0][*ia*]

< *timen*[i+1]. This function also linearly interpolates the E and H near zone data contained in *ehn* onto the *ehg*[0] reference grid at the *ia* grid point and loads these values into the next 6 columns of *ehg*. This routine loops from the *loc* input time point to the *ndata*-1 time point. Since both time grids, the input and the reference, are monotonically increasing, keeping track of the location of the previously found time point returned by the previous call, stored in *loc* , is faster than starting at the first location. The time step *delta_time* is used in the linear interpolation.

void **read_data_eh_xyz**(int **npoints*, int *ndata*, double **timeh*, double ****ehn*)

for small Cartesian problems, this reads in the data from the *npoints* near zone points in each of the six surfaces, creates the near zone time grid *timeh* and loads all for the E and H field data into the dynamically allocated *ehn*.

void **read_data_eh_rz**(int **npoints,* int *ndata*, int *nphi*, double **cosp*, double **sinp*, double **timeh*, double ****ehn*)

for small Cylindrical problems, reads in the *npoints* near zone data for each of the 3 near zone cylinder surfaces. Transforms from node to cell centered values, and converts these fields from cylindrical to Cartesian values, while creating the ϕ replicated values. For the bottom and top of the near zone cylinders, the z component is not required to calculated J and M in Eq. 6, so these are set to zero.

Temporary arrays *eh* and *ehold*, used to transform from node to cell centered values are created, used, and then deleted.

void **int_uw_ends**(int *ip*, double **cosp*, double **sinp*, int *ngrid*, int *is*, int *ic*, double **cellarea*, double ***ehg*, double ***uw*, double *cnst*)

for cylindrical problems, calculates the integral in Eqs. 4 and 5 without the time derivative by summing over the cell areas of each *ic* cell point on the bottom (*is* = 0) and the top (*is* =1) of the near zone cylinder creating the appropriate cross product using the E and H data contained in the time interpolated to the reference grid data stored in the *ehg* and using the previously stored cell areas stored. Each replicated i*p* ϕ data is used in the summation and the values are stored in the first six columns of *cosp* and *sinp* are used in the transformation from cylindrical to Cartesian coordinates and *cnst* is the constant in Eqs 4-5 with R set to 1. There are *ngrid* time points in each of the E and H field data.

void **int_uw_side**(int *ip*, double **cosp*, double **sinp*, int *ngrid*, int *is*, int *ic*, double **cellarea*, double ***ehg*, double ***uw*, double *cnst*)

for cylindrical problems, calculates the integral in Eqs. 4 and 5 without the time derivative by summing over the cell areas of each *ic* cell point on the side of the near zone cylinder creating the appropriate cross product using the E and H data contained in the time interpolated to the reference grid data stored in *ehg* and using the cell areas previously stored. Each replicated *ip* ϕ data is used in the summation and the values are stored in the first six columns of *uw* . *cosp* and *sinp* are used in the transformation from cylindrical to Cartesian coordinates and *cnst* is the constant in Eqs 4-5 with R set to 1. There are *ngrid* time points in each of the E and H field data.

long int **get_memsize**(void)

Creates a temporary file that contains the estimate of the amount of memory available for the problem obtained from the UNIX “free –b” command. This value is writtened to the temporary file named **XXXXABCD.txt** which is written in the execution directory. This large integer is then read, the temporary file deleted and the memory available in bytes returned as a long int.